

C# Generics

Eğer bir okulda veri yapıları dersi aldıysanız mutlaka bir stack veya queue örneği yapmışsınızdır. Stack örneğini ele alalım. Pop ve Push metodlarından oluşan Stack, LIFO(Last In First Out) prensibine göre çalışır. Kısacası listeye son giren(Push) eleman, ilk çıkar(Pop).

C# içerisinde bu işlemi yapan generic bir Stack sınıfı vardır. Öncelikle bu örnek üzerinden generic sınıfların önemini ve kullanım yerini anlatmaya çalışacağım.

Şimdi aşağıdaki örneği inceleyecek olursak, bir adet Stack sınıfının tanımlandığını göreceksiniz.

```
namespace ConsoleAppGenerics2
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<string> stack = new Stack<string>();
            stack.Push("Ankara");
            stack.Push("İstanbul");
            Console.WriteLine(stack.Pop());
            Console.ReadLine();
        }
    }
}
```

Stack sınıfı tanımlamasında <String> şeklinde bir ifade bu sınıfın generic bir yapıya sahip olduğunu gösterir. **Yani bu şu demek, ben “string” türüyle çalışıyorum! Eğer listenin int elemanlardan oluşmasını isteseydiniz, yapmanız gereken;**

`Stack<int> stack = new Stack<int>();` şeklinde bir tanımlama yapmak olacaktı. **Kısacası Generics sizin bir şablon oluşturup, o şablona göre istediğiniz veri türüyle çalışmanıza olanak sağlayan yapılardır.**

Şimdi başka bir örnekle olayı detaylandıralım. Bu örnekte, benim en çok kullandığım generic bir koleksiyonu kullanıyoruz. Yine <string> şeklinde bir tanımlama yaptığıma dikkat ediniz.

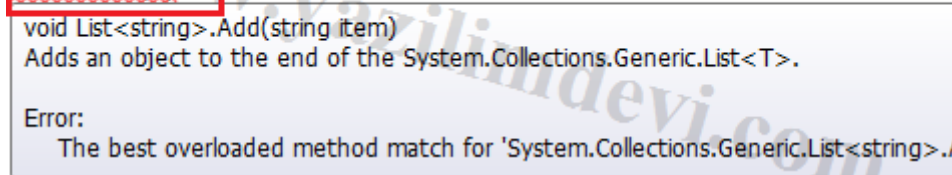
```
namespace ConsoleAppGenerics2
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> liste = new List<string>();
            liste.Add("Ankara");
            liste.Add("İstanbul");

            Console.WriteLine(liste[0]);
        }
    }
}
```

```
        Console.ReadLine();
    }
}
```

Eğer biz listeye int bir değer eklemeyi deneseydik, hata alırdık.

```
List<string> liste = new List<string>();
liste.Add("Ankara");
liste.Add(5);
```



Genericlerle çalışmak bir çok açıdan faydalıdır ama, en önemli özelliği nedir dersanız, ben **“Tip Güvenliği”** derim. Çalıştığınız veri türü, aynen sizin tanımladığınız gibi tutulur, farklı veri türlerine izin verilmez. Yani generic sınıflara benzer bir sınıf yazmak için **“object”** tipiyle çalışma devri kapanmış oluyor desem çok da abartmış olmam.(Bazı senaryolar var tabii kabul ediyorum.)

Genericler; Class,Struct,Interface,Delegate veya metod olabilirler. Şimdi herbirini tek tek örneklendireceğim.

Generic Sınıflar(Generic Classes)

Generic sınıflar, özellikle farklı veri tipleriyle çalışmak istediğinizde idealdir. Yukarıda üzerinde durduğum **“List”** koleksiyonu generic bir sınıftır. Biz hangi tip ile çalışmak istersek, herşey o tipe göre çalışır.

Generic bir sınıf tanımlamak çok kolaydır. Aşağıda ufak bir örnek görüyoruz. Yavaş yavaş biz de “List” benzeri bir generic sınıf yazacağız.

```
class MyList<T>
{
    //Kodlar
}
```

Burada <T> ifadesi sizin buraya herhangi bir tipi yazabileceğinizi belirtir. T harfi tamamen standartlarla alakalıdır. **“Type”** ifadesine karşılık gelir. Siz buraya herhangi bir harf veya ifade yazabilirsiniz.

Şimdi **“List”** benzeri tipimizi yazalım. Burada isteğim sizin algoritmaya değil sonuca odaklanmanızdır. Daha basit bir örnek yapmamamın nedeni syntax değil mantığı anlamaktır.

```
class MyList<T>
{
    T[] dizi;
    public void Ekle(T deger)
    {
```

```

try
{
    T[] geciciDizi = new T[dizi.Length];
    geciciDizi = dizi;
    dizi = new T[geciciDizi.Length + 1];
    for (int i = 0; i < geciciDizi.Length; i++)
    {
        dizi[i] = geciciDizi[i];
    }
    dizi[dizi.Length - 1] = deger;
}
catch
{
    dizi = new T[1];
    dizi[0] = deger;
}

}

public int ElemanSayisi
{
    get { return dizi.Length; }
}
}

```

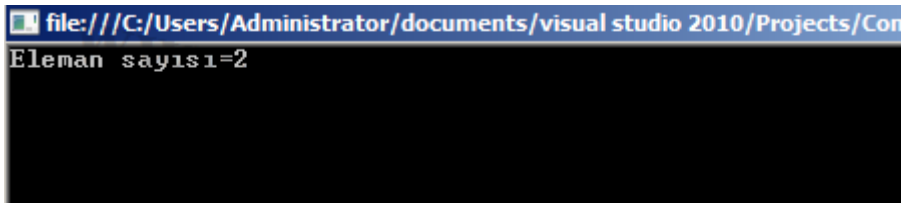
Bu örnekte bizim çalışacağımız tipe denk gelen her yer **T** harfiyle gösterilmiştir. Yani biz objeyi oluşturken hangi tipi seçeceğimiz konusunda özgürüz.

Şimdi bir kullanım örneği görelim.

```

class Program
{
    static void Main(string[] args)
    {
        MyList<string> liste = new MyList<string>();
        liste.Ekle("Engin Demiroğ");
        liste.Ekle("Yasemin Demiroğ");
        Console.WriteLine("Eleman sayısı={0}",liste.ElemanSayisi);
        Console.ReadLine();
    }
}

```



Birden fazla tip ile çalışmak

Yapmanız gereken tek işlem, çalışacağınız tipleri “,” ile ayırmaktır. Bu yapıya en uygun örnek “**Dictionary**” sınıfıdır.

```
Dictionary<  
class System.Collections.Generic.Dictionary<TKey, TValue>  
Represents a collection of keys and values.  
TKey: The type of the keys in the dictionary.
```

Aşağıda kendi yazdığım bir örneği görüyoruz.

```
class MyDictionary<T1, T2>  
{  
    //Kodlar  
}
```

Generic Constraints(Where Clauses)

Bazen yazdığınız genericleri sınırlamak isteyebilirsiniz. Örneğin YazilimDevi.Com projesini düşünelim.

- 1) Elimizde **Makale**, **Video** isimlerinde iki sınıf ve **Iders** isminde bir interface mevcut olsun.
- 2) Makale ve Video sınıfları Iders interface’ini inherit etsin.
- 3) Makale ya da Video işlemlerimizi yapan generic sınıfa bir kural koyalım.
- 4) Dolayısıyla T ile belirttiğimiz yere sadece Makale veya Video tipi gelebilsin.

Bu durumda ne yapacağız? Cevap, bir where ifadesi koymaktan ibarettir. Gelen sınıfın Iders interface’inden türemesi yeterlidir.

Aşağıda bu durumu örneklendirdim.

```
interface IDers  
{  
    void Listele();  
}  
  
class Makale:IDers  
{  
    public void Listele()  
    {  
        //Kodlar  
    }  
}  
  
class Video:IDers  
{  
    public void Listele()  
    {  
        //Kodlar  
    }  
}
```

```

    }

    class Islem<T>where T:IDers
    {
        //Kodlar
    }

```

Burada `where T:IDers` ifadesi, T'nin IDers interface'inden türetilmiş bir sınıf olması gerektiğini belirtir. Aşağıda, daha farklı kısıtlamalar ve açıklamalarını paylaşıyorum.

`class Islem<T>where T:Makale -> T sınıfı, Makale sınıfı veya Makale sınıfından türemiş bir sınıf olabilir.`

`class Islem<T>where T:class -> T, referans tipli olmalıdır.`

`class Islem<T>where T:struct -> T, değer tipli olmalıdır.`

`class Islem<T>where T:new() ->T,Parametresiz constructor bloğu olan herhangi bir tip olmalıdır.`

Eğer birden fazla tip için koşul koymak isterseniz aşağıdaki gibi bir kullanıma ihtiyacınız vardır.

```

class Islem<T1,T2>
    where T1:Makale
    where T2:Video
{
}

```

Generic Metodlar(Generic Methods)

Generic sınıflarda uyguladığımız yapıyı metodlar için de oluşturabiliriz. Aşağıda bir kaç farklı örnek paylaşıyorum.

```

class Islem
{
    public void Ekle<T>(T Ders)where T:IDers
    {
        //Kodlar
    }
    public List<TR> Listele<T1,TR>(T1 Ders)
    {
        List<TR> liste = new List<TR>();
        //Kodlar
        return liste;
    }
}

```

Generic “**Ekle**” metodumuz bizden “**T**” istiyor. Hemen sonrasında gelen “**where**” koşulu ise T, nin IDers interface’i veya IDers interface’ini inherit eden bir sınıf olma gereksinimidir.

Generic “**Listele**” metodumuz ise iki farklı tip istemekte olup, biri(TR) **Return** tipidir.

Generic Struct’lar(Generic Structs)

Her ne kadar çok fazla kullanmasak da, struct’ların, sınıfların aksine değer tipli olduğunu biliyoruz. Generic sınıflar gibi, generic struct’da yazılabilir. Bütün kurallar aynen geçerlidir. Ben yine de kendi oluşturduğumuz listeyi struct olarak ekliyorum.

```
struct MyList<T>
{
    T[] dizi;
    public void Ekle(T deger)
    {
        try
        {
            T[] geciciDizi = new T[dizi.Length];
            geciciDizi = dizi;
            dizi = new T[geciciDizi.Length + 1];
            for (int i = 0; i < geciciDizi.Length; i++)
            {
                dizi[i] = geciciDizi[i];
            }
            dizi[dizi.Length - 1] = deger;
        }
        catch
        {
            dizi = new T[1];
            dizi[0] = deger;
        }
    }

    public int ElemanSayisi
    {
        get { return dizi.Length; }
    }
}
```

Generic Delegates(Generic Delegates)

Delegeler, belli bir kalıba uyan metodları istediğiniz sırayla çalıştırmanızı sağlayan yapılardır. Doğal olarak generic olmayan bir delegate yazdığınız zaman, mecburen o kalıpta verdiğiniz metod imzalarına uymak zorundasınız. Generic delegate’ler yazarak farklı tiplerde metodlarla çalışabilme imkanı sağlayabiliriz. Aşağıda basit bir örnekle olayı detaylandırıyorum.

```
delegate void MyDelegate<T>(T deger);
```

```

class Islem
{
    public void DiziListele(string[] dizi)
    {
        Console.WriteLine("Diziyi Sıralıyorum...");
        foreach (var item in dizi)
        {
            Console.WriteLine(item);
        }
        Console.WriteLine();
    }

    public void DiziTersCevirListele(string[] dizi)
    {
        Array.Reverse(dizi);

        Console.WriteLine("Diziyi Tersten Sıralıyorum...");
        foreach (var item in dizi)
        {
            Console.WriteLine(item);
        }

        Console.WriteLine();
    }
}

class Program
{
    static void Main(string[] args)
    {
        Islem islem = new Islem();
        var delegem = new MyDelegate<string[]>(islem.DiziListele);
        delegem += islem.DiziTersCevirListele;

        string[] yazarlar=new string[3]{"Engin Demiroğ", "Veli
Kadir Kozan", "Ahmet Sait Duran"};
        delegem(yazarlar);
        Console.ReadLine();
    }
}

```

Generic Interfaces

Son olarak interface'lerin de generic kullanımlarına değinip konuyu kapatacağım.

```

interface IDers<T>
{
    void Ekle(T Ders);
}

```

```
class Makale
{ }

class Video
{ }

class Islem:IDers<Makale>,IDers<Video>
{
    public void Ekle(Makale Ders)
    {
        //Kodlar
    }

    public void Ekle(Video Ders)
    {
        //Kodlar
    }
}
```

IDers isimli interface, generic T tipiyle çalışıyor. Ekle metodunda da parametre olarak T tipini istiyor. Islem sınıfında, IDers interface'ini iki kere farklı tiplerle implement ediyoruz. Böylece hem az kod yazmış olduk, hem de rahatlıkla method overloading yapmış olduk.